Data-flow to Von Neumann : the SIGNAL approach

Paul Le Guernic, Thierry Gautier

UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 1229

Programme 1
Programmation, Calcul Symbolique
et Intelligence Artificielle

DATA-FLOW TO VON NEUMANN : THE SIGNAL APPROACH

Paul LE GUERNIC
Thierry GAUTIER

Mai 1990

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CÉDEX
FRANCE
Téléphone : 99 36 20 00
Télex : UNIRISA 950 473 F
Télécopie : 99 38 38 32

Data-flow to von Neumann: the SIGNAL approach

Paul Le Guernic Thierry Gautier

IRISA-INRIA
Campus de Beaulieu
35042 Rennes CEDEX
FRANCE

Publication Interne n° 531 - Avril 1990 - 22 Pages

Abstract

In real-time applications design, two parts have to be considered: such an application handles values (or more precisely, sequences of values), as any information system; moreover, (some notion of) time has an effect upon the behaviour of the application; not only scheduling but also valued results may depend upon the instants of events occurrences. In the traditional approaches, the time is a Daemon which unforeseeable behaviour the designer has to cope with. Conversely, the synchronous approach allows to partly look time as other values: the time is nothing but a partial order of events on which computing is possible. Merging synchronous and data-flow principles, SIGNAL has been defined from a small set of operators, for designing real-time programs running in (possibly) distributed environments. In this context, it is necessary to be able to generate a family of parallel processes (each process may be a sequential, a data-flow or a parallel process). This may be achieved by partitioning the initial nodes of a synchronized data-flow graph (at a micro-data-flow level) to build a new (macro-data-flow) graph such that, firstly, a sequential implementation may be locally calculated and, secondly, this implementation is correct whith respect to the logical time properties of the whole graph. In this paper, after a definition of SIGNAL kernel, we describe the graph and the synchronization space associated with each process; finally, properties of graphs, with respect to different scheduling policies, are studied in the framework of signal-graphs (graphs with distinguished input and output nodes).

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
(L. A. 227)
UNIVERSITÉ DE RENNES 1 I. N. S. A. DE RENNES

INSTITUT NATIONAL DE RECHERCHE
EN INFORMATIQUE ET EN AUTOMATIQUE
(LABORATOIRE DE RENNES)

D'une description flot de données vers une architecture von Neumann : l'approche SIGNAL

Résumé

La conception d'applications temps-réel nécessite de considérer les deux aspects suivants : une telle application manipule des valeurs (ou plus précisement, des suites de valeurs), comme tout système d'information, mais de plus, une certaine notion de temps a un effet sur le comportement de l'application ; l'ordonnancement d'une part, mais aussi les valeurs des résultats peuvent dépendre des instants d'occurrence des événements. Dans les approches traditionnelles, le temps est un "Démon" au comportement imprévisible auquel le concepteur doit s'accomoder. A l'inverse, l'approche synchrone permet de considérer en partie le temps comme les autres valeurs : le temps est défini comme un ordre partiel sur les événements, sur lequel il est possible d'effectuer des calculs. Mêlant les principes *synchrone* et *flot de données*, le langage SIGNAL est défini sur un ensemble réduit d'opérateurs permettant de programmer des applications temps-réel pouvant s'exécuter dans des environnements distribués. Dans ce contexte, il faut être capable d'engendrer une famille de processus parallèles, chaque processus pouvant être séquentiel, flot de données, ou parallèle. Ceci peut être obtenu en partitionnant l'ensemble des nœuds du *graphe flot de données synchronisé* (de niveau "micro-flot de données") de l'application, pour construire un nouveau graphe (de niveau "macro-flot de données") de telle sorte que, d'une part, une implémentation séquentielle puisse être calculée localement, et que, d'autre part, cette implémentation soit correcte vis-à-vis des propriétés temporelles du graphe complet. Dans ce papier, après une définition du noyau du langage SIGNAL, on décrit le graphe et l'espace des synchronisations associés à chaque processus ; enfin, les propriétés du graphe, relativement à diverses stratégies d'ordonnancement, sont étudiées dans le cadre d'un modèle de graphe permettant de distinguer les nœuds d'entrée et de sortie.[1]

[1] Ce papier a été présenté au *Workshop* "Dataflow: A status Report", Eilat (Israël), Mai 1989.

1 Introduction

The design and realization of real-time applications are a difficult challenge that numerous leading industrials have to take up. The availability of new tools allowing to apply well founded methods becomes actually urgent, mainly due to
- the multiplication of such industrial applications,
- the increasing cost of software development,
- its lifetime in front of fast evolution of technologies,
- and particularly, the use of these applications in very sensitive domains which require an absolute reliability (aeronautics, space, nuclear energy, military applications, etc.).

The complex behaviour of many real-time applications as well as the very large dimensions of some of them require specification tools which emancipate the designer from implementation problems. Moreover, because of the hardware progress, designing a real-time critical application bound to one particular technology may be just a nonsense. On the other hand, the highly demanding nature of these applications forces to consider as well the requirement of highly efficient and reliable implementation. In a lot of cases, the goal of high efficiency can be achieved by using parallel or distributed implementations. Furthermore, the environment of the application may constrain the designer to such distributed implementations.

This work should be seen in the context of development and implementation onto multiprocessor of real-time applications, from signal processing involving irregular control structures, to pure control of some *Discrete Event Dynamical Systems*. The first step is to bring out the key concepts for a high level specification of real-time applications. In these aplications, two parts have to be considered: such an application handles values (or more precisely, sequences of values), as any information system; moreover, (some notion of) time has an effect upon the behaviour of the application. Not only scheduling but also valued results may depend on the instants of events occurrences.

In the traditional asynchronous approaches[9, 11, 14], the time is a *Daemon* (as often as not, a disunited family of Daemons), whose unforeseeable behaviour the designer has to cope with. These approaches come up against serious difficulties: difficulty to test and prove programs (and consequently, difficulty to implement optimizations), difficulty to reuse programs in case of technological evolution. The main source of these problems is that any program has to take time to make calculations about time.

Conversely, the *synchronous* approach allows to reason about time as about any values domain. In this perspective, time must be considered along two of its three following characteristic aspects: partial order of events, simultaneity of events, and finally delays between events. In a synchronous framework, time is modelled as a chronology; durations are constraints to be verified at the implementation. Then it is possible to consider that calculations (and in particular calculations about the time) have a zero duration. This hypothesis is acceptable if any instruction of virtual zero duration has an effective bound duration. It is therefore possible to express recurrence only on the time.

This approach is that of *synchronous languages*. A first class of this kind of languages is rather automaton oriented; this is the case for the imperative language ESTEREL[4] and for the graphical STATECHARTS[10]. A second class consists of declarative languages where, as in LUCID[16], data flows are considered; this is the case for the languages LUSTRE[6] and SIGNAL[3, 8]. A SIGNAL process is a set of equations defining synchronized streams and the composition of such processes results in a new set of equations in a quite natural way.

The declarative form allows useful transformations of programs. In the framework of parallel implementation, so as to get a maximal freedom in order to distribute algorithms on processors, it is necessary to consider an *associative* and *commutative* composition operator. When designing a large and complex application, it is neither always possible nor suitable to handle all the temporal relations between events. As a consequence, each part of the application has to be defined with extra variables for synchronization purpose. Another approach is to describe constraints between the signals handled by this part and to have a compiler to synthesize a minimal scheduler. This is the basis of the SIGNAL approach. Merging synchronous and data-flow principles, SIGNAL has been defined from a small set of operators, to design real-time programs running in (possibly) distributed environments.

The compiler verifies the correction of the constraints by associating to each program a set of polynomial equations over the finite field $\mathbb{Z}/3\mathbb{Z}$. As a result of the resolution of these equations, it is possible to structure the set of instructions into a hierarchy based upon the events inclusion "frequences". The data-flow graph whose instructions are hierarchized is named the synchronized data-flow graph. To allow parallel implementation, this micro-data-flow graph is partitioned into a macro-data-flow graph verifying the two following rules:
- for each part, a sequential implementation may be locally calculated,
- this implementation is correct with respect to the logical time properties of the whole graph.

Then these macro-nodes are the atoms of the distribution. The sub-graph associated to each processor may have a sequential, a data-flow, or a parallel execution.

The language SIGNAL is presented in section 2. The establishing of the hierarchy is shown in section 3 through the use of synchronization space and of dependencies graph. Then, the graph partitioning policies are presented in section 4; this partitioning prepares a multiprocessor implementation.

2 Synchronous data-flow programming: the language SIGNAL

The language SIGNAL is defined from a small set of operators which constitute the kernel of the language. This allows to define simply formal semantics of the language and to derive new operators. The basis of the SIGNAL design is close to the *stream* (or *history*) concept such as defined in semantics of data-flow languages[12]. The main difference consists in the building of streams from *traces*; a trace is a stream in which the *bottom* ($\perp$) value (stating for "no event") may occur between two defined values. In the following, an informal presentation of SIGNAL timing operators motivates our work, related to traditional data-flow. More formal definitions are given in the appendix.

2.1 Synchronized data-flow

Consider as an example the following program expressed in some conventional data-flow language

$$\textbf{if } a > 0 \textbf{ then } x = a;\ y = x + a$$

What is the meaning of this program? In an interpretation where the arcs are considered as FIFO queues[1], if a is a sequence with non positive values, the queue associated with a

will grow forever, or (if a is a finite sequence) the queue associated with x will eventually be empty although a is non empty. It is not clear that the meaning of this program is the meaning that the author had in mind! Now, suppose that each FIFO queue consists of a single cell[7]. Then as soon as a negative value appears on the input, the execution can no longer go on: there is a deadlock. This is usually represented by the special undefined value $\perp$.

It would be somewhat significant if such deadlocks could be statically prevented. For that, it is necessary to be able to statically verify timing properties. Then the $\perp$ should be handled when reasoning about time, but it has to be considered with a non standard meaning. In the framework of synchronized data-flow, the $\perp$ will correspond to the absence of value at a given logical instant for a given variable (or *signal*). In particular, it must be possible to insert $\perp$'s between two defined values of a signal; such an insertion corresponds to some resynchronization of the signal. However, the main purpose of synchronized data-flow is that the whole synchronization should be completely handled at compile time, in such a way that the execution phase has nothing to do with $\perp$. This will be assumed by a static representation of the timing relations expressed by each operator. Syntactically, the timing will be implicit in the language. SIGNAL describes processes which communicate through (possibly infinite) sequences of (typed) values with implicit timing: the *signals*. For example, x denotes the infinite sequence $\{x_t\}_{t\geq 0}$ where the time index t is attached to this signal; signals defined with the same time index are said to have the same *clock*, so that clocks are equivalence classes of simultaneous signals. A SIGNAL program is made of a set of processes recursively composed from elementary processes; an elementary process is an expression defining one signal.

Consider a given operator which has for example two input signals and one output signal. We shall speak of *synchronous* signals if they are *logically* related in the following sense: for any t, the t^{th} token on the first input is evaluated with the t^{th} token on the second input, to produce the t^{th} token on the output. This is precisely the notion of *simultaneity*. However, for two tokens on a given signal, we can say that one is before the other (*chronology*). Then, for the synchronous approach, an *event* is a set of instantaneous calculations, or equivalently, of instantaneous communications.

2.2 Operators on signals

Each operator is presented informally; its syntax is described, the associated formal semantics is given in the appendix.

Monochronous processes

So as to ensure the right synchronizations between signals, the functions of the language (or, more generally, the relations) must be strongly synchronizing functions: at any instant, the t^{th} element of the result is defined from the t^{th} elements of the arguments; the function does not take time. Only one timing reference (or clock) is handled: such an operator defines a *monochronous* process.

(i) *Static* monochronous processes are defined by direct extension of instantaneous functions into functions acting on flows. Let f be a symbol which denotes a given n-ary function $[|f|]$ on instantaneous values (for example, arithmetic or boolean operation). Then, the

SIGNAL expression

$$a_{n+1} := f(a_1, \ldots, a_n)$$

defines an elementary process such that

$$\forall t \quad [|f|](a_1(t), \ldots, a_n(t)) = a_{n+1}(t)$$

where $a_i(t)$ denotes the t^{th} element of the sequence denoted by a_i. A byproduct of this definition is that all referred signals must be present simultaneously. Note that the definition expresses in fact a *relation* between values (and not really a function).

(ii) *Dynamic* monochronous processes come out when in a given timing reference, past values must be considered. The SIGNAL *delay* operator defines the signal whose t^{th} element is just the $(t-1)^{th}$ element of its input, at any instant but the first one, where it takes an initialization value (it may be compared with the data-flow D-box operator[1]):

$$a_2 := a_1 \ \$ \ v_0$$

At the first instant, the signal a_2 possesses the initialization value v_0. Then, at any instant, a_2 possesses the previous value of a_1. Here again, the involved signals must be present simultaneously.

Polychronous processes

Several timing references take place when SIGNAL operators corresponding to the data-flow *gates* and *merge*[1] are used. These operators define *polychronous* processes.

(i) **when**

The **when** operator corresponds to the data-flow *gate-if-true* (or T-gate). Like the T-gate, it has one data input and one boolean "control" input, but it has a precise meaning when one of the inputs holds $\perp$. In this case, the output is also $\perp$; at any logical instant where both input signals are defined, the output will be different from $\perp$ if and only if the control input holds the value *true*.

$$a_3 := a_1 \ \textbf{when} \ a_2$$

The **when** operator (or *condition*) allows to specify an undersampling relation through some boolean condition. The signal a_2 must be a boolean signal, then the signal a_3 is defined as an extraction of the signal a_1: a_3 has the value of a_1 each time the signal a_1 and the boolean a_2 are available, and moreover a_2 has the value *true*; otherwise, a_3 does not occur. The signal a_3 is less frequent than both a_1 and a_2.

(ii) **default**

To the data-flow merge (which needs a control input to select one data input) will correspond a deterministic merge with two data inputs, called **default**. The output will be defined (i.e., with a value different from $\perp$) at any logical instant where at less one of the inputs is defined (and non-defined otherwise); a priority makes it deterministic. We should say that the **default** is actually a data-flow operator though it is not strictly the case for

the conventional merge of data-flow schemas which is both data-flow and control-flow: the control input value of the merge determines which data input value is passed on the output arc.

$$a_3 := a_1 \textbf{ default } a_2$$

The **default** operator (deterministic *merge*) defines a_3 by merging a_1 and a_2, with priority to a_1 when both signals are present simultaneously: a_3 has the value of a_1 each time a_1 is available; it has the value of a_2 each time a_2 is available but a_1 is not; otherwise, a_3 does not occur. The signal a_3 is more frequent than both a_1 and a_2.

Composition

Resynchronizations (that is to say, possible insertions of $\perp$) have to take place when composing processes with common signals. However, this is only a formal manipulation. If $P1$ and $P2$ denote two processes, the *composition* of $P1$ and $P2$ defines a new process, denoted by

$$P1 \mid P2$$

where common names refer to common signals. Then, $P1$ and $P2$ communicate through their common signals.

Restriction

This operator allows to consider as local signals a subset of the signals defined in a given process. If $a_1, \ldots, a_n$ are signals defined in a process $P1$,

$$P1 \,/\, a_1, \ldots, a_n$$

defines a new process where communication ways (for composition) are those of $P1$, except $a_1, \ldots, a_n$.

Properties: The set of SIGNAL processes is a *commutative monoid*:

$$\begin{aligned} &(\mathbf{P1} \mid \mathbf{P2}) \mid \mathbf{P3} = \mathbf{P1} \mid (\mathbf{P2} \mid \mathbf{P3}) \\ &(\mathbf{P1} \mid \mathbf{P2}) = (\mathbf{P2} \mid \mathbf{P1}) \\ &\mathbf{P1} \mid \mathbf{1} = \mathbf{P1} \end{aligned}$$

where **1** is the process which has nor input neither output (and then never communicates).

2.3 A simple example

The purpose of the following process is to define a signal v which counts in the reverse order the number of occurrences of the events where a boolean signal *reset* holds the value *false*;

v is reinitialized (with a value $v0$) each time *reset* is *true*.

```
(| zv := v $ 0
 | vreset := v0 when reset
 | zvdec := zv when (not reset)
 | vdec := zvdec - 1
 | v := vreset default vdec
 | reach0 := true when (zv = 1)
 |) / zv, vreset, zvdec, vdec
```

Comments: v is defined with $v0$ each time *reset* is present and has the value *true* (operator **when**); otherwise (operator **default**), it takes the value of $zvdec - 1$, $zvdec$ being defined as the previous value of v (delay), zv, when this value is present and moreover, when *reset* is present and has the value *false* (operator **when**). The boolean signal *reach0* is defined (with the value *true*) when the previous value of v was equal to 1. Notice that v is decremented when *reset* has a value *false*.

This process has one input signal, *reset*, and two output signals, v and *reach0*.

2.4 A few words about the overall language

Derived operators are defined from the kernel of primitive operators, for example:

- **synchro** $a_1, \ldots, a_n$ used below specifies that $a_1, \ldots, a_n$ have the same clock.
- **event** a_1 delivers the "clock" of a_1 (i.e. an always *true* sequence).
- Some more complex operators (such as **if**, etc.) are also defined.

Program abstraction allows to declare a given process, together with its ways of communication, stated explicitly. As an example, the process of the previous section may be declared as

```
rcount  { integer v0
          ? logical reset
          ! logical reach0; integer v}
= (| zv := v $ 0
   | vreset := v0 when reset
   | zvdec := zv when (not reset)
   | vdec := zvdec - 1
   | v := vreset default vdec
   | reach0 := true when (zv = 1)
   |) / zv, vreset, zvdec, vdec
```

It may be referred to as, for example, *rcount*(10) ($v0$ is a formal parameter of the process; "?" stands as a tag for the input signals and "!" for the output ones).

3 Compiling SIGNAL programs

What are the relevant questions when compiling SIGNAL programs?
- Is the program deadlock free?
- Has it an effective execution?
- If so, what scheduling may be statically calculated (for a multiprocessor implementation)?
To be able to answer these questions, two basic tools are used before execution on a given architecture. The first one is the modelling of the synchronization relations in $\mathcal{F}_3$ by polynomials with coefficients in the finite field $\mathbb{Z}/3\mathbb{Z}$ of integers modulo 3. The second one is the directed graph of data dependencies.

3.1 The synchronization space

First, let us consider SIGNAL processes restricted to the single domain of boolean values. The condition

$$a_3 := a_1 \textbf{ when } a_2$$

expresses the following assertions:

- if a_1 is defined, and a_2 is defined and *true*, then a_3 is defined and $a_1 = a_3$
- if a_1 is not defined, or a_2 is not defined, or a_2 is defined and *false*, then a_3 is not defined

It appears that useful informations are (if a is a signal):

- a is defined and *false*
- a is defined and *true*
- a is not defined

They can be respectively encoded in the finite field $\mathbb{Z}/3\mathbb{Z}$ of integers modulo 3 as the following values:

$$-1 \quad 1 \quad 0$$

Then, if v is the encoding value associated to the signal a, the presence of the signal a may be clearly represented by $\mathbf{v}^2$. This representation of an indeterminate value of a (*true* or *false*) leads to an immediate generalization to non boolean values: their presence is encoded as 1 and their absence as 0. In this way, $\mathbf{a}^2$ may be considered as the proper clock of the signal a, where, to any signal a, a variable $\mathbf{a}$ (with the same notation) is associated in $\mathcal{F}_3$.

This principle is used to represent synchronization relations expressed through SIGNAL programs. The coding of the elementary operators is deduced from their definition. This coding is introduced below:

- To $a_{n+1} := f(a_1, \ldots, a_n)$ are associated the equations denoting the equality of the respective clocks of signals $a_{n+1}, a_1, \ldots, a_n$:

  $$\mathbf{a}_{n+1}^2 = \mathbf{a}_1^2 = \ldots = \mathbf{a}_n^2$$

  (all the synchronous processes are encoded in this way)

- Boolean relations may be completely encoded in $\mathcal{F}_3$. For instance,
  to $a_2 := \textbf{not}\ a_1$ corresponds $\mathbf{a}_2 = -\mathbf{a}_1$:
  if $a_1 = \mathit{true}$, then $\mathbf{a}_1 = 1$ and $-(\mathbf{a}_1) = -1$, which is associated to false.

- To $a_3 := a_1$ **when** a_2 (a_1, a_2, a_3 boolean signals) is associated the equation

$$\mathbf{a}_3 = \mathbf{a}_1(-\mathbf{a}_2 - \mathbf{a}_2^2)$$

  which may be interpreted as follows: a_3 holds the same value as a_1 ($\mathbf{a}_3 = \mathbf{a}_1$) when a_2 is true (when $-\mathbf{a}_2 - \mathbf{a}_2^2 = 1$).
  To $a_3 := a_1$ **when** a_2 (a_1, a_3 non boolean signals) is associated the equation

$$\mathbf{a}_3^2 = \mathbf{a}_1^2(-\mathbf{a}_2 - \mathbf{a}_2^2)$$

- To $a_3 := a_1$ **default** a_2 (a_1, a_2, a_3 boolean signals) is associated the equation

$$\mathbf{a}_3 = \mathbf{a}_1 + (1 - \mathbf{a}_1^2)\mathbf{a}_2$$

  which is interpreted as follows: a_3 has a value when a_1 is defined, i.e. when $\mathbf{a}_1^2 = 1$ (*then* a_3 holds the same value as a_1: $\mathbf{a}_3 = \mathbf{a}_1^2\mathbf{a}_1 = \mathbf{a}_1$), or when a_2 is defined but a_1 is not, i.e. when $(1 - \mathbf{a}_1^2)\mathbf{a}_2^2 = 1$ (then a_3 holds the same value as a_2: $\mathbf{a}_3 = (1 - \mathbf{a}_1^2)\mathbf{a}_2^2\mathbf{a}_2 = (1 - \mathbf{a}_1^2)\mathbf{a}_2$).
  To $a_3 := a_1$ **default** a_2 (a_1, a_2, a_3 non boolean signals) is associated the equation

$$\mathbf{a}_3^2 = \mathbf{a}_1^2 + (1 - \mathbf{a}_1^2)\mathbf{a}_2^2$$

Then the composition of SIGNAL processes collects the clocks expressions of every composing process.

The clock calculus

The algebraic coding of the synchronization relations has a double function. First, it is the way to detect synchronization errors. Consider for example the following program (which is that of §2.1):

$$c := a > 0 \mid x := a\ \textbf{when}\ c \mid y := x + a$$

The meaning of this program is "add a to (a **when** $a > 0$)"; remember that it must be rejected since the clocks are inconsistent. Its algebraic coding is

$$\begin{aligned} \mathbf{c}^2 &= \mathbf{a}^2 \\ \mathbf{x}^2 &= \mathbf{a}^2(-\mathbf{c} - \mathbf{c}^2) \\ \mathbf{y}^2 = \mathbf{x}^2 &= \mathbf{a}^2 \end{aligned}$$

which results in $\mathbf{c}^2 = \mathbf{a}^2 = \mathbf{y}^2 = \mathbf{x}^2 = \mathbf{a}^2(-\mathbf{c} - \mathbf{c}^2)$
and by substitution $\mathbf{c}^2 = \mathbf{c}^2(-\mathbf{c} - \mathbf{c}^2)$
and then $\mathbf{c} = 1$ or $\mathbf{c} = 0$.
But c is the result of the evaluation of the non boolean signal a. However the coding in $\mathcal{F}_3$ does not allow to reason about non boolean values, therefore the actual value (true or false) of c cannot be predicted and this value should not be constrained. Then the program

is rejected.

The other function of this coding is to organize the control of the program. An order relation may be defined on the set of clocks: a clock h^2 is said greater than a clock k^2, which is denoted by $h^2 \geq k^2$, if the set of instants of k is included in the set of instants of h (k is an undersampling of h). The set of clocks with this relation is a lattice. The purpose of the clock calculus is to synthesize the upper bound of the lattice, which is called the *master clock*, and to define each clock by a calculus expression, i.e. an undersampling of the master clock according to values of boolean signals. However, for a given SIGNAL process, the master clock may not be the clock of a signal of the process. In this case, several maxima (local master clocks) will be found.

For a program to be correct, the partial order induced by the inclusion of instants, restricted to the undersamplings by a boolean condition, must be a tree. This is necessary to obtain a deterministic program. The root of the tree is the more frequent clock. Moreover, any clock expression may be recursively reduced to a sum of monomials, where each monomial is a product of undersamplings (otherwise, the clock is a root).

An example

Consider again the process *rcount* of §2.4. The clock calculus finds the following clocks:

$$\begin{gathered}
\mathtt{reset}^2 \\
\mathtt{vreset}^2 = -\mathtt{reset} - \mathtt{reset}^2 \\
\mathtt{v}^2 = \mathtt{zv}^2 = \alpha^2 = (-\mathtt{reset} - \mathtt{reset}^2) + (\mathtt{reset} - \mathtt{reset}^2)\mathtt{v}^2 \\
\mathtt{vdec}^2 = \mathtt{zvdec}^2 = \mathtt{v}^2(\mathtt{reset} - \mathtt{reset}^2) \\
\mathtt{reach0}^2 = -\alpha - \mathtt{v}^2
\end{gathered}$$

where α is the coding of $zv = 1$.

The clock calculus do not synthesize a master clock for this process. In fact, it is non deterministic: when *reset* is *false*, then *zvdec* is defined if *zv* is defined, i.e. if *v* is defined; but *v* is defined (when *reset* is *false*) if *vdec* is defined, i.e. if *zvdec* is defined, and then, when *reset* is *false*, an occurrence of *v* may occur, but not necessarily occurs.
The hierarchy is represented by the following SIGNAL process, which defines several trees

(whose roots are *h_6_h*, *h_9_h* and *h_13_h*):

```
(| (| synchro h_6_h, reset
   | (| h_7_h := true when reset
      | synchro h_7_h, vreset
      | h_8_h := true when (not reset)
      |)
   |)
 | (| h_9_h := h_8_h when h_13_h
   | synchro h_9_h, vdec, zvdec
   |)
 | (| h_13_h := h_7_h default h_9_h
   | synchro h_13_h, v, zv
   | (| h_14_h := true when (zv = 1)
      | synchro h_14_h, reach0
      |)
   |)
 |)
```

The hierarchy is represented by the composition embeddings; the *h_i_h*'s represent the names of the clocks considered as signals (the number *i* is given by the compiler); the **synchro** processes associate a given clock with the signals available at this clock.

Now, we consider the following process, where *rcount* is used in some context

```
usercount  { integer v0
           ? logical h
           ! logical reach0; integer v}
= (| synchro h, v
   | reset := ((event reach0) when (event h)) default (not (event h))
   | rcount(v0)
   |) / reset
```

An external clock *h* defines the instants at which *v* has a value. The *reset* signal is also synchronous with *h* and it has the value *true* exactly when *reach*0 is present.

There is a master clock ($\mathbf{h}^2 = \mathbf{v}^2 = \mathbf{reset}^2$) and a tree may be built by the compiler.

3.2 The graph of conditional dependencies

The second tool necessary to implement a SIGNAL program on a given architecture is the graph of data dependencies. Then, according to criteria to be developed, it will be possible to define sub-graphs that may be distributed on different processors. However, a classical data-flow graph would not really represent the data dependencies of a SIGNAL program. Since the language handles signals whose clocks may be different, the dependencies are not constant. For that reason, the graph has to express *conditional* dependencies, where the

conditions are nothing but the clocks at which dependencies are effective. Moreover, in addition to dependencies between signals, the following relation has to be considered: for any signal a, the values of a cannot be known before its clock; in other words, a depends on $\mathbf{a}^2$. This relation will be implicit below.

The *Conditional Dependencies Graph* calculated by the SIGNAL compiler for a given program is a labelled directed graph where
- the vertices are the signals, plus clock variables,
- the arcs represent dependencies relations,
- the labels are polynomials on $\mathcal{F}_3$ which represent the clocks at which the relations are valid.

The following describes the dependencies associated to elementary processes. The notation $x^2 : a_1 \rightarrow a_2$ means that a_2 depends on a_1 exactly when $x^2 = 1$. It has to be noticed that the processes which involve only boolean signals do not generate data dependencies. Then, we consider only processes defining non boolean signals:

$$a_{n+1} := f(a_1, \ldots, a_n) \quad \mathbf{a}^2_{n+1} : a_1 \rightarrow a_{n+1}, \ldots, \mathbf{a}^2_{n+1} : a_n \rightarrow a_{n+1}$$

$$a_3 := a_1 \textbf{ when } a_2 \qquad \mathbf{a}^2_3 : a_1 \rightarrow a_3, \quad \mathbf{a}^2_3 : a_2 \rightarrow \mathbf{a}^2_3$$

$$a_3 := a_1 \textbf{ default } a_2 \qquad \mathbf{a}^2_1 : a_1 \rightarrow a_3, \quad \mathbf{a}^2_2 - \mathbf{a}^2_1\mathbf{a}^2_3 : a_2 \rightarrow a_3$$

Notice that the delay does not produce data dependencies (nevertheless, remember that any signal is preceded by its clock).

The graph, together with the clock calculus, is used to detect incorrect dependencies. Such a bad dependency will appear as a circuit in the graph. However, since dependencies are labelled by clocks, some circuits may not occur at any time. An effective circuit is such that the product of the labels of its arcs is not null. This may be compared with the cycle sum test of [15], to detect deadlock on the dependency graph of a data-flow program.

4 Towards multiprocessor implementation

The efficiency of any application relies on its implementation. Of course, methods for implementing programs onto multiprocessors depend on the architecture of these multiprocessors (but they must be as general as possible). As an alternative to the von Neumann model, new architectures are proposed, and, for instance, data-flow machines begin to appear. However, our purpose is not here to discuss the appropriateness of such architectures (see [2]). In the field of real-time applications, specific constraints may be crucial: particularly, it is necessary to minimize synchronizations and transfer costs. In this context, we rather consider here general MIMD multiprocessors.

4.1 The principle: hierarchization of the graph

The two objects generated by the SIGNAL compiler from a given program, namely, the hierarchy of the clocks and the dependencies graph, are not independent. They are in fact the internal representation of the program and constitute the basis for studying multiprocessor implementation. Consider a SIGNAL process rewritten according to the hierarchy of its clocks. The calculations of the signals available at a given clock are put together in the sub-tree corresponding to this clock. In the same way, we consider the dependencies

sub-graphs associated to a given clock.

In order to reduce the complexity of an automaton implementing a given SIGNAL program when it has to be distributed, the graphs themselves may be partitioned, so as to obtain some abstraction of sets of calculations. The following defines some criteria that could be applied (see [13] for a complete presentation).

4.2 Calculi on the graph

Implementing parts on different processors may require to group together or at the contrary to break up some of them. Then the composition of graphs has to be defined precisely, and particularly, it is necessary to distinguish the communication ways of a given dependencies graph. For that, we consider graphs with distinguished input and output nodes, the *signal-graphs*. For the sake of simplicity, we work in a first step on *monochronous* sub-graphs: we do not consider the clock labels (this does not restrict the whole program to be monochronous). The following definitions may be generalized to handle conditional sub-graphs.

4.2.1 A model

Signal-graph: A signal-graph $\mathcal{G}(P)$ associated to a process P is a quadruple $G = (N, \Gamma, I, O)$ where

- N is a finite set of nodes, each of which represents the expression defining a signal,
- $\Gamma \subset N \times N$ is the set of dependencies between the expressions as stated in §3.2,
- $I \subset N$ is the set of (visible) input nodes,
- $O \subset N$ is the set of visible output nodes.

The Conditional Dependencies Graph is a signal-graph associated with a set Σ of reduced polynomials expressions on $\mathcal{F}_3$ and two mappings

$$h_N : N \rightarrow \Sigma$$
$$h_\Gamma : \Gamma \rightarrow \Sigma$$

representing the clocks of signals and dependencies (sea §3.2). Two signals have the same clock if and only if they are associated to equivalent polynomials. In the following, we are concerned with partitions of equivalent (with respect to equality of clocks) signals and arcs. So we will only consider signal-graphs.

A distributed implementation of a process P is defined by a set of communicating sub-processes of P in the following expression: $P_1|\ldots|P_n$. Each process P_i is executed on one processor; it is represented by a sub-signal-graph $\mathcal{G}(P_i)$ of the signal-graph $G = \mathcal{G}(P)$ associated to P: to a partition of a SIGNAL process corresponds such a set of sub-signal-graphs. Each sub-graph, say $G_i = (N_i, \Gamma_i, I_i, O_i)$, is such that for any cut arc (x, y) in G where x is in G_i and y is in G_j, then a node (x, y) is added in G_i and G_j; moreover, Γ_i contains $(x, (x, y))$ and Γ_j contains $((x, y), y)$ as dependencies. Any node (respectively, any arc) in G_i is a node (respectively, an arc) from G or is added as described above. The sets N and Γ are partitioned in subsets of (N_i) and (Γ_i).

A partition of a signal-graph into sub-signal-graphs must be such that the composition

of the sub-signal-graphs is the initial signal-graph. The composition $G_i \cdot G_j$ of sub-signal-graphs G_i and G_j is defined by the union of nodes and arcs of G_i and G_j, with the following special rule:
if $(x,(x,y))$ is an arc of G_i and $((x,y),y)$ is an arc of G_j, then the arc (x,y) is added in the composed graph and the node (x,y) is deleted.
It is easy to sea that $\mathcal{G}(P_1) \cdot \ldots \cdot \mathcal{G}(P_n) = \mathcal{G}(P_1|\ldots|P_n)$.

4.2.2 Scheduling policies

Our goal is to define locally a *scheduling* for each sub-graph, which does not create any deadlock whatever the scheduling of other sub-graphs is. So we are interested in circuit-free composition of "scheduled" graphs.

For a given signal-graph G, we consider the preorder over N obtained by the reflexive-transitive closure of Γ: it is denoted by $\leq_G$. A scheduling is nothing but a partial order compatible with $\leq_G$. A good scheduling will be obtained by a *reinforcement* of the initial graph such that the initial order is respected: this reinforcement must not create circuits. The goal of the reinforcement is to construct sequential parts in the graph.

We call *scheduling-graph* of a signal-graph $G = (N,\Gamma,I,O)$, a circuit-free signal-graph $G_S = (N,\Gamma_S,I_S,O)$ such that $\Gamma^* \subset \Gamma_S^*$.

In order to obtain a given number n of communicating sequential processes, it is interesting to consider the scheduling-graphs for which there exists a partition into n *segments* (that is to say, sets of nodes connected by an elementary path): they are called *n-scheduling-graphs*.

The problem is to find the sub-graphs of a given graph, for which a scheduling-graph can be calculated, and such that the composition of the scheduling-graphs of all the sub-graphs is a scheduling-graph of the whole graph. Moreover, it is important to consider the scheduling-graphs G_S of a signal-graph G which are *circuit-consistent*, i.e., for any signal-graph G', if $G_S \cdot G'$ has a circuit, then $G \cdot G'$ has a circuit.

Consider pairs of nodes (x,y) of a sub-graph G_i such that x is an input of the sub-graph, y is an output, and there is no path from x to y. Then the value of x in the overall graph may need the value of y to be calculated. So a local scheduling where x is input before y is output would be non consistent: it generates a deadlock. Nevertheless, for many structures of graphs it is possible to build a sequential scheduling. This results from the following observation: if an input a precedes a set of outputs which is strictly included in a set of outputs preceded by an input b, then we can insure that (in a circuit-free graph) a cannot participate to the elaboration of b. So input of b can always occur before input of a. It is this idea which motivates the definition of a *semi-granule*.

We will say that two inputs in a sub-graph G_i are equivalent if they precede the same set of outputs in G_i; we denote $\equiv_i$ this relation. Considering the partition (with p parts) $I_i/\equiv_i$ of inputs, we associate to each part $\hat{m}_j$ the set n_j of outputs that are preceded by $\hat{m}_j$. Then if there exists a strictly decreasing sequence (for inclusion) $n_1,\ldots,n_p$ of such defined subsets of outputs, a scheduling in which $\hat{m}_j \rightarrow (n_j - n_{j+1}) \rightarrow \hat{m}_{j+1}$ is circuit-consistent. We name *semi-granules* these sub-graphs (a signal-graph is a semi-granule if and only if it has a circuit-consistent 1-scheduling-graph).

If $I_i/\equiv_i$ is a singleton, then we may have any scheduling greater than the Γ_i^* relation. We name *granules* the sub-graphs verifying this property. A granule is a signal-graph such that there exists a path from any of its inputs to any of its outputs; any scheduling-graph of a granule is circuit-consistent. We present below an algorithm to build granules.

4.3 An algorithm

A quasi-linear algorithm of partitioning into granules has been defined. It builds a granule from one of the nodes of the graph, called germ, by absorbing next nodes if the granularity property is preserved. As soon as a granule cannot become bigger, the algorithm builds another one from a new germ.

Let G_1 and G_2 be two already built granules (maybe reduced to one node). If G_2 is the only successor of G_1, then we say that G_1 is a *root* of G_2. If G_1 is the only predecessor of G_2, then we say that G_2 is a *bough* on G_1.
The algorithm simply uses the following rules (based on local properties):
- the composition of a granule G with a bough on G is a granule
- the composition of a granule G with a root of G is a granule.

The other configurations do not give rise to granules.

5 Conclusion

We have presented the data-flow synchronous language SIGNAL and a methodology to implement SIGNAL programs onto multiprocessor architectures. However, defining partitions over the graphs is only a first step in order to distribute tasks on a multiprocessor. And maybe, it would not be realistic to envisage an optimal automatic distribution. Then, tools will be supplied to the users, for example to group together partitions on some processor. For that, the syntactic view of a program at any stage of its compilation is essential. In particular, such tools could be graphical tools[5]. A version of the compiler is currently available: it produces sequential FORTRAN code for a whole program. An implementation onto a multi-transputer architecture is under development, it will produce OCCAM code. Then, for a given implementation, it is possible to generate a SIGNAL program which performs the temporal simulation of the implementation. A granule or a group of granules is a sub-process of a SIGNAL process representing a processor; physical links between processors are also represented as processes. Then, each calculation instruction and each link-process may be automatically replaced by appropriate operations on integers (maximum and addition of constant), which represent the time consumption of the different actions.
SIGNAL has already been used successfully to program realistic examples: speech recognition, targets detection, etc.

References

[1] Arvind and K. P. Gostelow. Some relationships between asynchronous interpreters of a dataflow language. In E. J. Neuhold, editor, *Formal Description of Programming Concepts*, pages 95–119, North-Holland, 1978.

[2] Arvind and R. A. Iannucci. Two fundamental issues in multiprocessing. In S. S. Thakkar, editor, *Selected Reprints on Dataflow and Reduction Architectures*, pages 140–164, Computer Society Press of the IEEE, 1987.

[3] A. Benveniste, B. Le Goff, and P. Le Guernic. *Hybrid Dynamical Systems theory and the language* SIGNAL. Research Report 838, INRIA, Rocquencourt, April 1988.

[4] G. Berry and L. Cosserat. The ESTEREL synchronous programming language and its mathematical semantics. In S. D. Brookes, A. W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency*, pages 389–448, Lecture Notes in Computer Science, 197, Springer-Verlag, 1985.

[5] P. Bournai, V. Kerscaven, and P. Le Guernic. Un environnement graphique pour la conception d'applications temps réel. In *Colloque sur l'ingénierie des interfaces homme-machine*, pages 181–190, Sophia-Antipolis, 1989.

[6] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: a declarative language for programming synchronous systems. In *14th ACM Symposium on Principles of Programming Languages*, pages 178–188, Munich, 1987.

[7] J. B. Dennis, J. B. Fossen, and J. P. Linderman. Data flow schemas. In A. Ershov and V. A. Nepomniaschy, editors, *International Symposium on Theoretical Programming*, pages 187–216, Lecture Notes in Computer Science, 5, Springer-Verlag, 1974.

[8] T. Gautier, P. Le Guernic, and L. Besnard. SIGNAL: a declarative language for synchronous programming of real-time systems. In G. Kahn, editor, *Functional programming languages and computer architecture*, pages 257–277, Lecture Notes in Computer Science, 274, Springer-Verlag, 1987.

[9] N. Gehani. ADA *Concurrent Programming*. Prentice-Hall; 1984.

[10] D. Harel. STATECHARTS: a visual formalism for complex systems". *Science of Computer Programming*, 8(3):231–274, June 1987.

[11] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[12] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing 74*, pages 471–475, North-Holland, 1974.

[13] B. Le Goff. *Inférence de contrôle hiérarchique : application au temps-réel*. PhD thesis, Université de Rennes 1, 1989.

[14] A. Roscoe and C. A. R. Hoare. *The Laws of* OCCAM *Programming. Technical Monograph PRG-53*, Oxford University Computing Laboratory, 1986.

[15] W. W. Wadge. An extensional treatment of dataflow deadlock. In G. Kahn, editor, *Semantics of Concurrent Computation*, pages 285–299, Lecture Notes in Computer Science, 70, Springer-Verlag, 1979.

[16] W. W. Wadge and E. A. Ashcroft. LUCID, *the Dataflow Programming Language*. Academic Press, 1985.

Appendix: formal definition of SIGNAL

The appendix contains a formalization of the basis of synchronized data-flow Signals will be represented by *streams*.

A The set of traces

Let us consider a finite set $A = \{a_1, \ldots, a_n\}$ of typed variables called *ports*.
We define the following notations:
For each $a_i \in A$, $\mathcal{D}_{a_i}$ is the domain of values (integers, reals, booleans...) that may be held by a_i at every instant. We set

$$\mathcal{D} = \bigcup_{i=1}^{n} \mathcal{D}_{a_i}$$
$$\mathcal{D}^{\perp} = \mathcal{D} \cup \{\perp\}$$

where $\perp \notin \mathcal{D}$ denotes the absence of value associated to a port at a given instant.
$\mathcal{D}_{a_i}^{\perp}$ and $\mathcal{D}_{A1}^{\perp}$ are defined in the same way ($A1$ is a subset of A).

For every non empty subset $A1$ of A we consider

- the set of the applications m defined from $A1$ into $\mathcal{D}_{A1}^{\perp}$, called set of *events* on $A1$ and denoted by $\mathcal{E}_{A1}$.
  $m(a) = \perp$ means: a does not hold a value.
  $m(a) = v$ means: a holds the value v.
  $m(A1) = \{x/a \in A1, m(a) = x\}$

  $$\mathcal{E}_{A1} = A1 \rightarrow \mathcal{D}_{A1}^{\perp}$$

  is the set of events on $A1$.

  $$\mathcal{E} = \bigcup_{A1 \subset A} \mathcal{E}_{A1}$$

  is the set of all possible events.
  $\mathcal{E}_{\emptyset} = \{\emptyset\}$ is the event on an empty set of ports.

- the set of the applications T defined from the set $\mathbf{N}$ of natural integers into $\mathcal{E}_{A1}$, called set of *traces* on $A1$ and denoted by $\mathcal{T}_{A1}^{\perp}$:

  $$\mathcal{T}_{A1}^{\perp} = \mathbf{N} \rightarrow \mathcal{E}_{A1}$$

  The set of all possible traces is

  $$\mathcal{T}^{\perp} = (\bigcup_{A1 \subset A} \mathcal{T}_{A1}^{\perp})$$

  Moreover, we set

  $$\mathcal{T}_{\emptyset} = \mathbf{1} = \mathbf{N} \rightarrow \mathcal{E}_{\emptyset}$$
  $$\mathbf{0} = \mathbf{N} \rightarrow (A \rightarrow \{\perp\})$$

- Restriction on $A2 \subset A1$:
  for $A2 \subset A1$ T being defined on $A1$
  $\forall t$ $A2.T(t)$ is the restriction of $T(t)$ to $A2$.

$$A2.T \quad \mathbf{N} \to T_{A2}^{\perp}$$
$$\forall t, \forall a \in A2 \; A2.T(t)(a) = T(t)(a)$$

  $\emptyset.T \in T_{\emptyset}$ (which is a singleton)

- Streams on $A1 \subset A$:
  we call *stream* on $A1$ any trace T of $T_{A1}^{\perp}$ such that

$$\exists t \; ((T(t)(A1) = \{\perp\}) \Rightarrow \forall s \geq t \; (T(s)(A1) = \{\perp\}))$$

  We denote by T (respectively, T_{A1}) the set of streams of $T^{\perp}$ (respectively, $T_{A1}^{\perp}$).

$$0_T \quad \mathbf{N} \to (A \to \{\perp\})$$

  $\emptyset.T$ is a stream.

- The temporal expansion of a trace (insertion of $\perp$'s).
  Let $f1$ be a strictly increasing application from $\mathbf{N} \to \mathbf{N}$ and $T1$ a trace on $A1$.
  $T: \mathbf{N} \to A1 \to \mathcal{D}_{A1}^{\perp}$, $A1 \neq \emptyset$ is a trace on $A1$. We call *expansion* of T by $f1$ the trace $f1 \times T : \mathbf{N} \to A1 \to \mathcal{D}_{A1}^{\perp}$ defined by

$$(f1 \times T) \circ f1 = T$$
$$\forall t \; \forall s \quad f1(t) < s < f1(t+1) \quad (f1 \times T)(s)(A1) = \{\perp\}$$

  The function $f1$ is called expansion function.
  $f1 \cdot T_{\emptyset} = T_{\emptyset}$

B Definition of the operators

The semantics associated to each operator is defined as some set of streams. Formally a SIGNAL process on $A1 \subseteq A$ is a set of streams on $A1$ (that is to say, a subset of T_{A1}). It is defined from basic operators, with the composition, by *constraints* on streams.

Monochronous processes

Static monochronous processes

The SIGNAL expression

$$a_{n+1} := f(a_1, \ldots, a_n)$$

will be denoted by P:

$$P \equiv a_{n+1} := f(a_1, \ldots, a_n)$$

Then we associate to the expression P the process $\mathbf{P}$ defined by

$$\mathbf{P} = \{T \in T_{a_1, \ldots, a_{n+1}} / \forall t \quad v_{n+1} = [[f]](v_1, \ldots, v_n), v_i = t(\{a_i\}.T)(t) \text{ or } T(t)(a_i) = \{\perp\}\}$$

(ii) Dynamic monochronous processes: the delay

$$P \equiv a_2 := a_1 \$ v_0$$

$$\mathbf{P} = \{T \in \mathcal{T}_{\{a_1,a_2\}} / \forall t > 0 : (\{a_2\}.T)(t) = (\{a_1\}.T)(t-1)) \; and \; (\{a_2\}.T)(0) = v_0\}$$

Polychronous processes

(i) **when**

$$P \equiv a_3 := a_1 \textbf{ when } a_2$$

$$\begin{aligned}\mathbf{P} = \{T \in \mathcal{T}_{\{a_1,a_2,a_3\}} / \; \forall t \quad & (\{a_2\}.T)(t) = true \Rightarrow (\{a_3\}.T(t) = (\{a_1\}.T)(t) \\ & (\{a_2\}.T)(t) \neq true \Rightarrow (\{a_3\}.T)(t) = \bot\}\end{aligned}$$

(ii) **default**

$$P \equiv a_3 := a_1 \textbf{ default } a_2$$

$$\begin{aligned}\mathbf{P} = \{T \in \mathcal{T}_{\{a_1,a_2,a_3\}} / \; \forall t \quad & (\{a_1\}.T)(t) \neq \bot \Rightarrow (\{a_3\}.T)(t) = (\{a_1\}.T)(t) \\ & (\{a_1\}.T)(t) = \bot \Rightarrow (\{a_3\}.T)(t) = (\{a_2\}.T)(t)\}\end{aligned}$$

Composition

$$P \equiv P1 \mid P2$$

$\mathbf{P1} \subset \mathcal{T}_{A1}$, $\mathbf{P2} \subset \mathcal{T}_{A2}$

$$\begin{aligned}\mathbf{P} = \{T \in \mathcal{T}_{A1 \cup A2} / \; & \exists\, T1 \in \mathbf{P1},\ \exists\, T2 \in \mathbf{P2}, \\ & \exists\, f1, f2 \; expansion \; functions \\ & A1.T = f1 \times T1, A2.T = f2 \times T2\}\end{aligned}$$

Restriction

$$P \equiv P1 \;/\; a_1, \ldots, a_n$$

$\mathbf{P1} \subset \mathcal{T}_{A1}$

$$\mathbf{P} = \{T \in \mathcal{T}_{A1-\{a_1,\ldots,a_n\}} / \exists\, T1 \in \mathbf{P1} \; A1 - \{a_1, \ldots, a_n\}.T1 = T\}$$

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique

ISSN 0249-6399